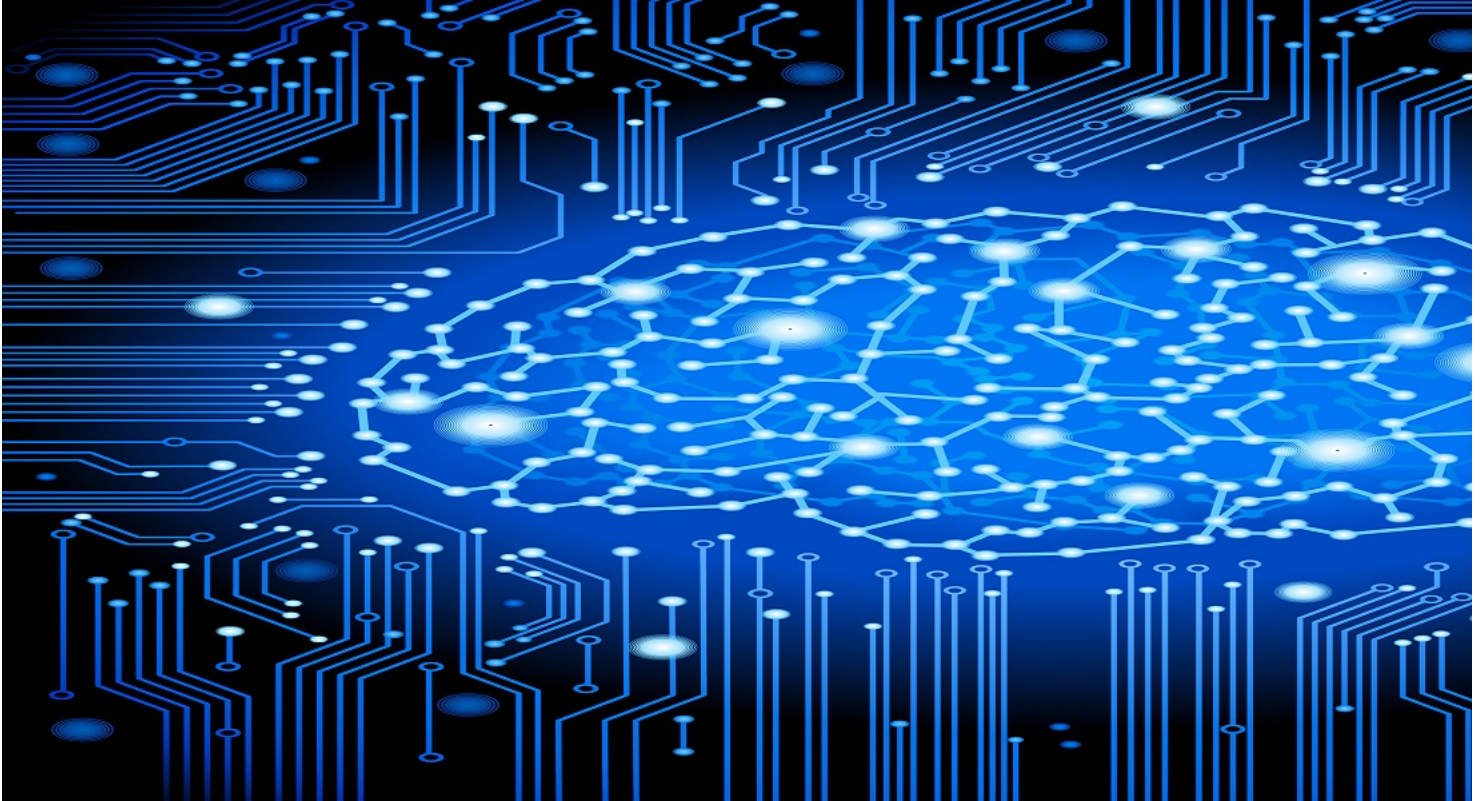


[\[关闭\]](#)

@hanbingtao 2017-08-29 23:41 字数 14413 阅读 28556

# 零基础入门深度学习(7) - 递归神经网络

机器学习 深度学习入门



无论即将到来的是大数据时代还是人工智能时代，亦或是传统行业使用人工智能在云上处理大数据的时代，作为一个有理想有追求的程序员，不懂深度学习（Deep Learning）这个超热的技术，会不会感觉马上就out了？现在救命稻草来了，《零基础入门深度学习》系列文章旨在讲帮助爱编程的你从零基础达到入门级水平。零基础意味着你不需要太多的数学知识，只要会写程序就行了，没错，这是专门为程序员写的文章。虽然文中会有很多公式你也许看不懂，但同时也会有更多的代码，程序员的你一定能看懂的（我周围是一群狂热的Clean Code程序员，所以我写的代码也不会很差）。

## 文章列表

[零基础入门深度学习\(1\) - 感知器](#)[零基础入门深度学习\(2\) - 线性单元和梯度下降](#)[零基础入门深度学习\(3\) - 神经网络和反向传播算法](#)[零基础入门深度学习\(4\) - 卷积神经网络](#)[零基础入门深度学习\(5\) - 循环神经网络](#)[零基础入门深度学习\(6\) - 长短期记忆网络\(LSTM\)](#)[零基础入门深度学习\(7\) - 递归神经网络](#)

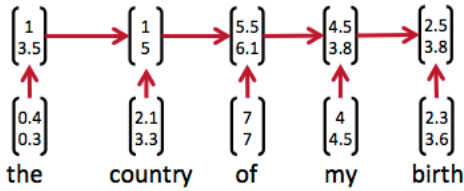
## 往期回顾

在前面的文章中，我们介绍了**循环神经网络**，它可以用来处理包含序列结构的信息。然而，除此之外，信息往往还存在着诸如树结构、图结构等更复杂的结构。对于这种复杂的结构，**循环神经网络**就无能为力了。本文介绍一种更

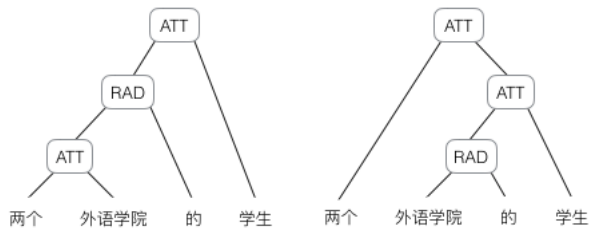
为强大、复杂的神经网络：**递归神经网络 (Recursive Neural Network, RNN)**，以及它的训练算法**BPTS (Back Propagation Through Structure)**。顾名思义，**递归神经网络**（巧合的是，它的缩写和**循环神经网络**一样，也是RNN）可以处理诸如树、图这样的**递归结构**。在文章的最后，我们将实现一个**递归神经网络**，并介绍它的几个应用场景。

递归神经网络是啥

因为神经网络的输入层单元个数是固定的，因此必须用循环或者递归的方式来处理长度可变的输入。**循环神经网络**实现了前者，通过将长度不定的输入分割为等长度的小块，然后再依次的输入到网络中，从而实现了神经网络对变长输入的处理。一个典型的例子是，当我们处理一句话的时候，我们可以把一句话看作是词组成的序列，然后，每次向**循环神经网络**输入一个词，如此循环直至整句话输入完毕，**循环神经网络**将产生对应的输出。如此，我们就能处理任意长度的句子了。入下图所示：

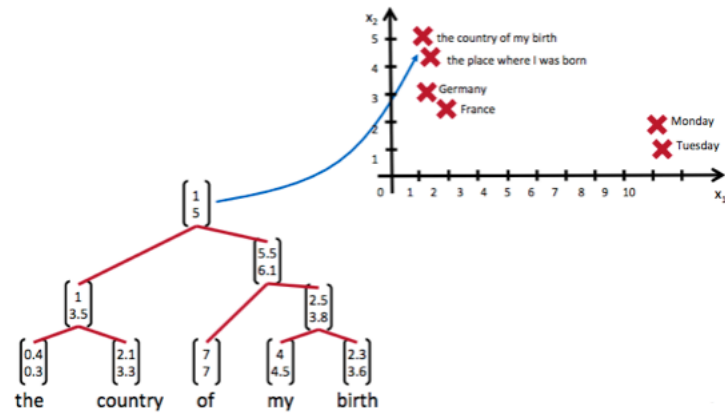


然而，有时候把句子看做是词的序列是不够的，比如下面这句话『两个外语学院的学生』：



上图显示了这句话的两个不同的语法解析树。可以看出来这句话有歧义，不同的语法解析树则对应了不同的意思。一个是『两个外语学院的/学生』，也就是学生可能有许多，但他们来自于两所外语学校；另一个是『两个/外语学院的学生』，也就是只有两个学生，他们是外语学院的。为了能够让模型区分出两个不同的意思，我们的模型必须能够按照树结构去处理信息，而不是序列，这就是**递归神经网络**的作用。当面对按照树/图结构处理信息更有效的任务时，**递归神经网络**通常都会获得不错的结果。

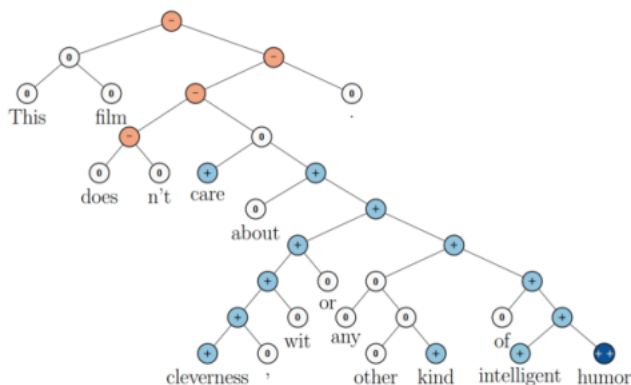
**递归神经网络**可以把一个树/图结构信息编码为一个向量，也就是把信息映射到一个语义向量空间中。这个语义向量空间满足某类性质，比如语义相似的向量距离更近。也就是说，如果两句话（尽管内容不同）它的意思是相似的，那么把它们分别编码后的两个向量的距离也相近；反之，如果两句话的意思截然不同，那么编码后向量的距离则很远。如下图所示：



从上图我们可以看到，**递归神经网络**将所有的词、句都映射到一个2维向量空间中。句子『the country of my birth』和句子『the place where I was born』的意思是接近的，所以表示它们的两个向量在向量空间中的距离很近。另外两

个词『Germany』和『France』因为表示的都是地点，它们的向量与上面两句话的向量的距离，就比另外两个表示时间的词『Monday』和『Tuesday』的向量的距离近得多。这样，通过向量的距离，就得到了一种语义的表示。

上图还显示了自然语言**可组合**的性质：词可以组成句、句可以组成段落、段落可以组成篇章，而更高层的语义取决于底层的语义以及它们的组合方式。**递归神经网络**是一种表示学习，它可以将词、句、段、篇按照他们的语义映射到同一个向量空间中，也就是把可组合（树/图结构）的信息表示为一个个有意义的向量。比如上面这个例子，**递归神经网络**把句子"the country of my birth"表示为二维向量[1,5]。有了这个『编码器』之后，我们就可以以这些有意义的向量为基础去完成更高级的任务（比如情感分析等）。如下图所示，**递归神经网络**在做情感分析时，可以比较好的处理否定句，这是胜过其他一些模型的：



在上图中，蓝色表示正面评价，红色表示负面评价。每个节点是一个向量，这个向量表达了以它为根的子树的情感评价。比如"intelligent humor"是正面评价，而"care about cleverness wit or any other kind of intelligent humor"是中性评价。我们可以看到，模型能够正确的处理doesn't的含义，将正面评价转变为负面评价。

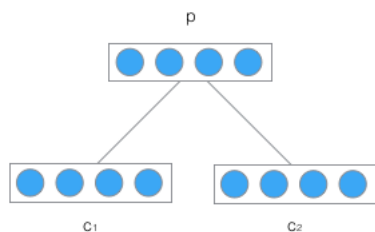
尽管**递归神经网络**具有更为强大的表示能力，但是在实际应用中并不太流行。其中一个主要原因是，**递归神经网络**的输入是树/图结构，而这种结构需要花费很多人工去标注。想象一下，如果我们用**循环神经网络**处理句子，那么我们可以直接把句子作为输入。然而，如果我们用**递归神经网络**处理句子，我们就必须把每个句子标注为语法解析树的形式，这无疑要花费非常大的精力。很多时候，相对于**递归神经网络**能够带来的性能提升，这个投入是不太划算的。

我们已经基本了解了**递归神经网络**是做什么用的，接下来，我们将探讨它的算法细节。

## 递归神经网络的前向计算

接下来，我们详细介绍一下**递归神经网络**是如何处理树/图结构的信息的。在这里，我们以处理树型信息为例进行介绍。

**递归神经网络**的输入是两个子节点（也可以是多个），输出就是将这两个子节点编码后产生的父节点，父节点的维度和每个子节点是相同的。如下图所示：

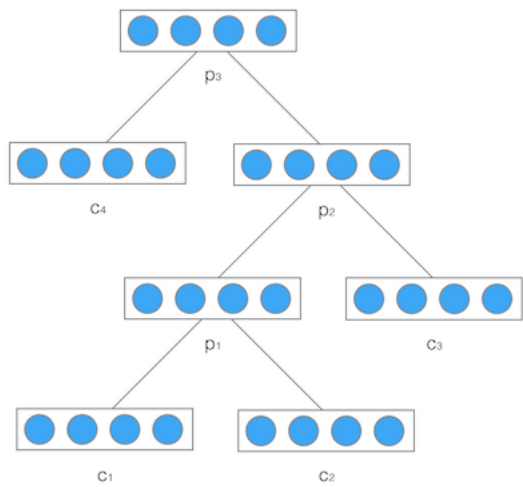


$c_1$ 和 $c_2$ 分别是表示两个子节点的向量， $p$ 是表示父节点的向量。子节点和父节点组成一个全连接神经网络，也就是子节点的每个神经元都和父节点的每个神经元两两相连。我们用矩阵 $W$ 表示这些连接上的权重，它的维度将是 $d \times 2d$ ，其中， $d$ 表示每个节点的维度。父节点的计算公式可以写成：

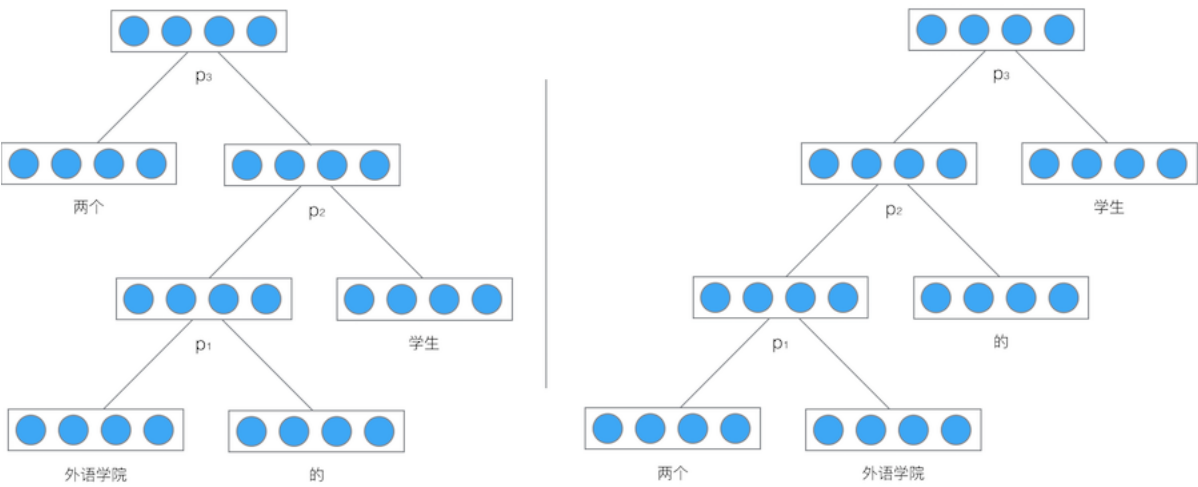
$$p = \tanh(W \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + b) \quad (\text{式1})$$

在上式中， $\tanh$ 是激活函数（当然也可以用其它的激活函数）， $\mathbf{b}$ 是偏置项，它也是一个维度为 $d$ 的向量。如果读过前面的文章，相信大家已经非常熟悉这些计算了，在此不做过多的解释了。

然后，我们把产生的父节点的向量和子节点的向量再次作为网络的输入，再次产生它们的父节点。如此递归下去，直至整棵树处理完毕。最终，我们将得到根节点的向量，我们可以认为它是对整棵树的表示，这样我们就实现了把树映射为一个向量。在下图中，我们使用递归神经网络处理一棵树，最终得到的向量 $\mathbf{p}_3$ ，就是对整棵树的表示：



举个例子，我们使用递归神经网络将『两个外语学校的学生』映射为一个向量，如下图所示：



最后得到的向量 $\mathbf{p}_3$ 就是对整个句子『两个外语学校的学生』的表示。由于整个结构是递归的，不仅仅是根节点，事实上每个节点都是以其为根的子树的表示。比如，在左边的这棵树中，向量 $\mathbf{p}_2$ 是短语『外语学院的学生』的表示，而向量 $\mathbf{p}_1$ 是短语『外语学院的』的表示。

式1就是递归神经网络的前向计算算法。它和全连接神经网络的计算没有什么区别，只是在输入的过程中需要根据输入的树结构依次输入每个子节点。

需要特别注意的是，递归神经网络的权重 $\mathbf{W}$ 和偏置项 $\mathbf{b}$ 在所有的节点都是共享的。

## 递归神经网络的训练

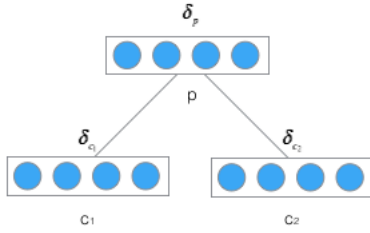
递归神经网络的训练算法和循环神经网络类似，两者不同之处在于，前者需要将残差 $\delta$ 从根节点反向传播到各个子节点，而后者是将残差 $\delta$ 从当前时刻 $t_k$ 反向传播到初始时刻 $t_1$ 。

下面，我们介绍适用于递归神经网络的训练算法，也就是BPTS算法。



## 误差项的传递

首先，我们先推导将误差从父节点传递到子节点的公式，如下图：



定义 $\delta_p$ 为误差函数 $E$ 相对于父节点 $p$ 的加权输入 $\mathbf{net}_p$ 的导数，即：

$$\delta_p \stackrel{\text{def}}{=} \frac{\partial E}{\partial \mathbf{net}_p}$$

设 $\mathbf{net}_p$ 是父节点的加权输入，则

$$\mathbf{net}_p = W \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix} + \mathbf{b}$$

在上述式子里， $\mathbf{net}_p$ 、 $\mathbf{c}_1$ 、 $\mathbf{c}_2$ 都是向量，而 $W$ 是矩阵。为了看清楚它们的关系，我们将其展开：

$$\begin{bmatrix} \mathbf{net}_{p_1} \\ \mathbf{net}_{p_2} \\ \dots \\ \mathbf{net}_{p_n} \end{bmatrix} = \begin{bmatrix} w_{p_1 c_{11}} & w_{p_1 c_{12}} & \dots & w_{p_1 c_{1n}} & w_{p_1 c_{21}} & w_{p_1 c_{22}} & \dots & w_{p_1 c_{2n}} \\ w_{p_2 c_{11}} & w_{p_2 c_{12}} & \dots & w_{p_2 c_{1n}} & w_{p_2 c_{21}} & w_{p_2 c_{22}} & \dots & w_{p_2 c_{2n}} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w_{p_n c_{11}} & w_{p_n c_{12}} & \dots & w_{p_n c_{1n}} & w_{p_n c_{21}} & w_{p_n c_{22}} & \dots & w_{p_n c_{2n}} \end{bmatrix} \begin{bmatrix} c_{11} \\ c_{12} \\ \dots \\ c_{1n} \\ c_{21} \\ c_{22} \\ \dots \\ c_{2n} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} \quad (1)$$

在上面的公式中， $p_i$ 表示父节点 $p$ 的第 $i$ 个分量； $c_{1i}$ 表示 $c_1$ 子节点的第 $i$ 个分量； $c_{2i}$ 表示 $c_2$ 子节点的第 $i$ 个分量； $w_{p_i c_{jk}}$ 表示子节点 $c_j$ 的第 $k$ 个分量到父节点 $p$ 的第 $i$ 个分量的权重。根据上面展开后的矩阵乘法形式，我们不难看出，对于子节点 $c_{jk}$ 来说，它会影响父节点所有的分量。因此，我们求误差函数 $E$ 对 $c_{jk}$ 的导数时，必须用到全导数公式，也就是：

$$\frac{\partial E}{\partial c_{jk}} = \sum_i \frac{\partial E}{\partial \mathbf{net}_{p_i}} \frac{\partial \mathbf{net}_{p_i}}{\partial c_{jk}} \quad (2)$$

$$= \sum_i \delta_{p_i} w_{p_i c_{jk}} \quad (3)$$

有了上式，我们就可以把它表示为矩阵形式，从而得到一个向量化表达：

$$\frac{\partial E}{\partial \mathbf{c}_j} = U_j \delta_p \quad (4)$$

其中，矩阵 $U_j$ 是从矩阵 $W$ 中提取部分元素组成的矩阵。其单元为：

$$u_{jk} = w_{p_k c_{ji}}$$

上式看上去可能会让人晕菜，从下图，我们可以直观的看到 $U_j$ 到底是啥。首先我们把 $W$ 矩阵拆分为两个矩阵 $W_1$ 和 $W_2$ ，如下图所示：

$$W = \begin{bmatrix} W_1 & W_2 \end{bmatrix}$$

$$W_1 = \begin{bmatrix} w_{p_1 c_{11}} & w_{p_1 c_{12}} & \dots & w_{p_1 c_{1n}} \\ w_{p_2 c_{11}} & w_{p_2 c_{12}} & \dots & w_{p_2 c_{1n}} \\ \dots & \dots & \dots & \dots \\ w_{p_n c_{11}} & w_{p_n c_{12}} & \dots & w_{p_n c_{1n}} \end{bmatrix}$$

$$W_2 = \begin{bmatrix} w_{p_1 c_{21}} & w_{p_1 c_{22}} & \dots & w_{p_1 c_{2n}} \\ w_{p_2 c_{21}} & w_{p_2 c_{22}} & \dots & w_{p_2 c_{2n}} \\ \dots & \dots & \dots & \dots \\ w_{p_n c_{21}} & w_{p_n c_{22}} & \dots & w_{p_n c_{2n}} \end{bmatrix}$$

显然，子矩阵 $W_1$ 和 $W_2$ 分别对应子节点 $c_1$ 和 $c_2$ 的到父节点 $p$ 权重。则矩阵 $U_j$ 为：

$$U_j = W_j^T$$

也就是说，将误差项反向传递到相应子节点 $c_j$ 的矩阵 $U_j$ 就是其对应权重矩阵 $W_j$ 的转置。

现在，我们设 $\text{net}_{c_j}$ 是子节点 $c_j$ 的加权输入， $f$ 是子节点 $c$ 的激活函数，则：

$$c_j = f(\text{net}_{c_j}) \quad (5)$$

这样，我们得到：

$$\delta_{c_j} = \frac{\partial E}{\partial \text{net}_{c_j}} \quad (6)$$

$$= \frac{\partial E}{\partial c_j} \frac{\partial c_j}{\partial \text{net}_{c_j}} \quad (7)$$

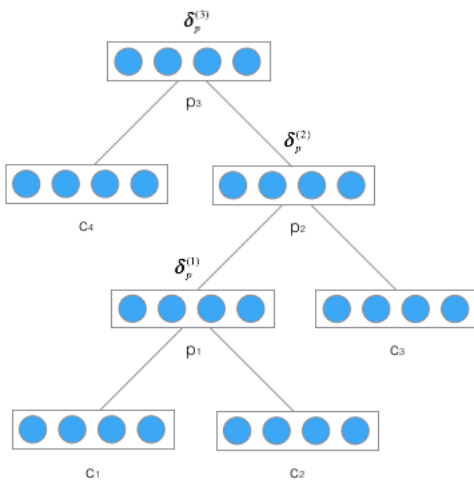
$$= W_j^T \delta_p \circ f'(\text{net}_{c_j}) \quad (8)$$

如果我们将不同子节点 $c_j$ 对应的误差项 $\delta_{c_j}$ 连接成一个向量 $\delta_c = \begin{bmatrix} \delta_{c_1} \\ \delta_{c_2} \end{bmatrix}$ 。那么，上式可以写成：

$$\delta_c = W^T \delta_p \circ f'(\text{net}_c) \quad (\text{式2})$$

式2就是将误差项从父节点传递到其子节点的公式。注意，上式中的 $\text{net}_c$ 也是将两个子节点的加权输入 $\text{net}_{c_1}$ 和 $\text{net}_{c_2}$ 连在一起的向量。

有了传递一层的公式，我们就不难写出逐层传递的公式。



上图是在树型结构中反向传递误差项的全景图，反复应用式2，在已知 $\delta_p^{(3)}$ 的情况下，我们不难算出 $\delta_p^{(1)}$ 为：

$$\delta^{(2)} = W^T \delta_p^{(3)} \circ f'(\mathbf{net}^{(2)}) \quad (9)$$

$$\delta_p^{(2)} = [\delta^{(2)}]_p \quad (10)$$

$$\delta^{(1)} = W^T \delta_p^{(2)} \circ f'(\mathbf{net}^{(1)}) \quad (11)$$

$$\delta_p^{(1)} = [\delta^{(1)}]_p \quad (12)$$

在上面的公式中， $\delta^{(2)} = \begin{bmatrix} \delta_c^{(2)} \\ \delta_p^{(2)} \end{bmatrix}$ ， $[\delta^{(2)}]_p$ 表示取向量 $\delta^{(2)}$ 属于节点p的部分。

## 权重梯度的计算

根据加权输入的计算公式：

$$\mathbf{net}_p^{(l)} = W\mathbf{c}^{(l)} + \mathbf{b}$$

其中， $\mathbf{net}_p^{(l)}$ 表示第l层的父节点的加权输入， $\mathbf{c}^{(l)}$ 表示第l层的子节点。 $W$ 是权重矩阵， $\mathbf{b}$ 是偏置项。将其展开可得：

$$\mathbf{net}_{p_j}^l = \sum_i w_{ji} c_i^l + b_j$$

那么，我们可以求得误差函数在第l层对权重的梯度为：

$$\frac{\partial E}{\partial w_{ji}^{(l)}} = \frac{\partial E}{\partial \mathbf{net}_{p_j}^{(l)}} \frac{\partial \mathbf{net}_{p_j}^{(l)}}{\partial w_{ji}^{(l)}} \quad (13)$$

$$= \delta_{p_j}^{(l)} c_i^{(l)} \quad (14)$$

上式是针对一个权重项 $w_{ji}$ 的公式，现在需要把它扩展为对所有的权重项的公式。我们可以把上式写成矩阵的形式（在下面的公式中， $m=2n$ ）：

$$\frac{\partial E}{\partial W^{(l)}} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}^{(l)}} & \frac{\partial E}{\partial w_{12}^{(l)}} & \cdots & \frac{\partial E}{\partial w_{1m}^{(l)}} \\ \frac{\partial E}{\partial w_{21}^{(l)}} & \frac{\partial E}{\partial w_{22}^{(l)}} & \cdots & \frac{\partial E}{\partial w_{2m}^{(l)}} \\ \vdots & & & \end{bmatrix} \quad (15)$$

$$= \begin{bmatrix} \frac{\partial E}{\partial w_{n1}^{(l)}} & \frac{\partial E}{\partial w_{n2}^{(l)}} & \cdots & \frac{\partial E}{\partial w_{nm}^{(l)}} \\ \delta_{p_1}^{(l)} c_1^l & \delta_{p_1}^{(l)} c_2^l & \cdots & \delta_{p_1}^{(l)} c_m^l \\ \delta_{p_2}^{(l)} c_1^l & \delta_{p_2}^{(l)} c_2^l & \cdots & \delta_{p_2}^{(l)} c_m^l \\ \vdots & & & \end{bmatrix} \quad (16)$$

$$= \delta^{(l)} (\mathbf{c}^{(l)})^T \quad (\text{式3}) \quad (17)$$

式3就是第l层权重项的梯度计算公式。我们知道，由于权重 $W$ 是在所有层共享的，所以和循环神经网络一样，递归神经网络的最终的权重梯度是各个层权重梯度之和。即：

$$\frac{\partial E}{\partial W} = \sum_l \frac{\partial E}{\partial W^{(l)}} \quad (\text{式4})$$

因为循环神经网络的证明过程已经在[零基础入门深度学习\(4\) - 卷积神经网络](#)一文中给出，因此，递归神经网络『为什么最终梯度是各层梯度之和』的证明就留给读者自行完成啦。

接下来，我们求偏置项**b**的梯度计算公式。先计算误差函数对第l层偏置项**b<sup>(l)</sup>**的梯度：

$$\frac{\partial E}{\partial b_j^{(l)}} = \frac{\partial E}{\partial \text{net}_{p_j}^{(l)}} \frac{\partial \text{net}_{p_j}^{(l)}}{\partial b_j^{(l)}} \quad (18)$$

$$= \delta_{p_j}^{(l)} \quad (19)$$

把上式扩展为矩阵的形式：

$$\frac{\partial E}{\partial \mathbf{b}^{(l)}} = \begin{bmatrix} \frac{\partial E}{\partial b_1^{(l)}} \\ \frac{\partial E}{\partial b_2^{(l)}} \\ \dots \\ \frac{\partial E}{\partial b_n^{(l)}} \end{bmatrix} \quad (20)$$

$$= \begin{bmatrix} \delta_{p_1}^{(l)} \\ \delta_{p_2}^{(l)} \\ \dots \\ \delta_{p_n}^{(l)} \end{bmatrix} \quad (21)$$

$$= \delta_p^{(l)} \quad (\text{式5}) \quad (22)$$

式5是第l层偏置项的梯度，那么最终的偏置项梯度是各个层偏置项梯度之和，即：

$$\frac{\partial E}{\partial \mathbf{b}} = \sum_l \frac{\partial E}{\partial \mathbf{b}^{(l)}} \quad (\text{式6})$$

## 权重更新

如果使用梯度下降优化算法，那么权重更新公式为：

$$\mathbf{W} \leftarrow \mathbf{W} + \eta \frac{\partial E}{\partial \mathbf{W}}$$

其中， $\eta$ 是学习速率常数。把式4带入到上式，即可完成权重的更新。同理，偏置项的更新公式为：

$$\mathbf{b} \leftarrow \mathbf{b} + \eta \frac{\partial E}{\partial \mathbf{b}}$$

把式6带入到上式，即可完成偏置项的更新。

这就是递归神经网络的训练算法BPTS。由于我们有了前面几篇文章的基础，相信读者们理解BPTS算法也会比较容易。

## 递归神经网络的实现

完整代码请参考GitHub: [https://github.com/hanbt/learn\\_dl/blob/master/recursive.py](https://github.com/hanbt/learn_dl/blob/master/recursive.py) (python2.7)



现在，我们实现一个处理树型结构的**递归神经网络**。

在文件的开头，加入如下代码：

```
1.  #!/usr/bin/env python
2.  # -*- coding: UTF-8 -*-
3.
4.
5.  import numpy as np
6.  from cnn import IdentityActivator
```

上述四行代码非常简单，没有什么需要解释的。IdentityActivator激活函数是在我们介绍**卷积神经网络**时写的，现在引用一下它。

我们首先定义一个树节点结构，这样，我们就可以用它保存卷积神经网络生成的整棵树：

```
1.  class TreeNode(object):
2.      def __init__(self, data, children=[], children_data=[]):
3.          self.parent = None
4.          self.children = children
5.          self.children_data = children_data
6.          self.data = data
7.          for child in children:
8.              child.parent = self
```

接下来，我们把**递归神经网络**的实现代码都放在RecursiveLayer类中，下面是这个类的构造函数：

```
1.  # 递归神经网络实现
2.  class RecursiveLayer(object):
3.      def __init__(self, node_width, child_count,
4.                  activator, learning_rate):
5.          ...
6.          递归神经网络构造函数
7.          node_width: 表示每个节点的向量的维度
8.          child_count: 每个父节点有几个子节点
9.          activator: 激活函数对象
10.         learning_rate: 梯度下降算法学习率
11.         ...
12.         self.node_width = node_width
13.         self.child_count = child_count
14.         self.activator = activator
15.         self.learning_rate = learning_rate
16.         # 权重数组W
17.         self.W = np.random.uniform(-1e-4, 1e-4,
18.                                     (node_width, node_width * child_count))
19.         # 偏置项b
20.         self.b = np.zeros((node_width, 1))
21.         # 递归神经网络生成的树的根节点
22.         self.root = None
```

下面是前向计算的实现：

```
1.         def forward(self, *children):
2.             ...
3.             前向计算
4.             ...
5.             children_data = self.concatenate(children)
6.             parent_data = self.activator.forward(
7.                 np.dot(self.W, children_data) + self.b
8.             )
9.             self.root = TreeNode(parent_data, children
10.                                , children_data)
```

forward函数接收一系列的树节点对象作为输入，然后，**递归神经网络**将这些树节点作为子节点，并计算它们的父节点。最后，将计算的父节点保存在self.root变量中。

上面用到的concatenate函数，是将各个子节点中的数据拼接成一个长向量，其代码如下：

```

1.     def concatenate(self, tree_nodes):
2.         '''
3.         将各个树节点中的数据拼接成一个长向量
4.         '''
5.         concat = np.zeros((0,1))
6.         for node in tree_nodes:
7.             concat = np.concatenate((concat, node.data))
8.         return concat

```

下面是反向传播算法BPTS的实现：

```

1.     def backward(self, parent_delta):
2.         '''
3.         BPTS反向传播算法
4.         '''
5.         self.calc_delta(parent_delta, self.root)
6.         self.W_grad, self.b_grad = self.calc_gradient(self.root)
7.
8.     def calc_delta(self, parent_delta, parent):
9.         '''
10.        计算每个节点的delta
11.        '''
12.        parent.delta = parent_delta
13.        if parent.children:
14.            # 根据式2计算每个子节点的delta
15.            children_delta = np.dot(self.W.T, parent_delta) * (
16.                self.activator.backward(parent.children_data)
17.            )
18.            # slices = [(子节点编号, 子节点delta起始位置, 子节点delta结束位置)]
19.            slices = [(i, i * self.node_width,
20.                       (i + 1) * self.node_width)
21.                      for i in range(self.child_count)]
22.            # 针对每个子节点, 递归调用calc_delta函数
23.            for s in slices:
24.                self.calc_delta(children_delta[s[1]:s[2]],
25.                                parent.children[s[0]])
26.
27.    def calc_gradient(self, parent):
28.        '''
29.        计算每个节点权重的梯度, 并将它们求和, 得到最终的梯度
30.        '''
31.        W_grad = np.zeros((self.node_width,
32.                           self.node_width * self.child_count))
33.        b_grad = np.zeros((self.node_width, 1))
34.        if not parent.children:
35.            return W_grad, b_grad
36.        parent.W_grad = np.dot(parent.delta, parent.children_data.T)
37.        parent.b_grad = parent.delta
38.        W_grad += parent.W_grad
39.        b_grad += parent.b_grad
40.        for child in parent.children:
41.            W, b = self.calc_gradient(child)
42.            W_grad += W
43.            b_grad += b
44.        return W_grad, b_grad

```

在上述算法中，calc\_delta函数和calc\_gradient函数分别计算各个节点的误差项 $\delta$ 以及最终的梯度。它们都采用递归算法，先序遍历整个树，并逐一完成每个节点的计算。

下面是梯度下降算法的实现（没有weight decay），这个非常简单：

```

1.     def update(self):
2.         '''
3.         使用SGD算法更新权重
4.         '''
5.         self.W -= self.learning_rate * self.W_grad

```

```
6.         self.b -= self.learning_rate * self.b_grad
```

以上就是递归神经网络的实现，总共100行左右，和上一篇文章的LSTM相比简单多了。

最后，我们用梯度检查来验证程序的正确性：

```
1. def gradient_check():
2.     '''
3.     梯度检查
4.     '''
5.     # 设计一个误差函数，取所有节点输出项之和
6.     error_function = lambda o: o.sum()
7.
8.     rnn = RecursiveLayer(2, 2, IdentityActivator(), 1e-3)
9.
10.    # 计算forward值
11.    x, d = data_set()
12.    rnn.forward(x[0], x[1])
13.    rnn.forward(rnn.root, x[2])
14.
15.    # 求取sensitivity map
16.    sensitivity_array = np.ones((rnn.node_width, 1),
17.                                dtype=np.float64)
18.
19.    # 计算梯度
20.    rnn.backward(sensitivity_array)
21.
22.    # 检查梯度
23.    epsilon = 10e-4
24.    for i in range(rnn.W.shape[0]):
25.        for j in range(rnn.W.shape[1]):
26.            rnn.W[i,j] += epsilon
27.            rnn.reset_state()
28.            rnn.forward(x[0], x[1])
29.            rnn.forward(rnn.root, x[2])
30.            err1 = error_function(rnn.root.data)
31.            rnn.W[i,j] -= 2*epsilon
32.            rnn.reset_state()
33.            rnn.forward(x[0], x[1])
34.            rnn.forward(rnn.root, x[2])
35.            err2 = error_function(rnn.root.data)
36.            expect_grad = (err1 - err2) / (2 * epsilon)
37.            rnn.W[i,j] += epsilon
38.            print 'weights(%d,%d): expected - actual %.4e - %.4e' % (
39.                i, j, expect_grad, rnn.W_grad[i,j])
40.
41.    return rnn
```

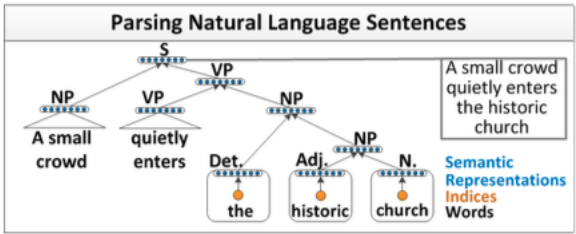
下面是梯度检查的结果，完全正确，OH YEAH !

```
>>> rnn = recursive.gradient_check()
weights(0,0): expected - actual 3.8925e-05 - 3.8925e-05
weights(0,1): expected - actual -3.9918e-04 - -3.9918e-04
weights(0,2): expected - actual 4.9996e+00 - 4.9996e+00
weights(0,3): expected - actual 5.9995e+00 - 5.9995e+00
weights(1,0): expected - actual 8.0841e-05 - 8.0841e-05
weights(1,1): expected - actual -3.1535e-04 - -3.1535e-04
weights(1,2): expected - actual 4.9997e+00 - 4.9997e+00
weights(1,3): expected - actual 5.9996e+00 - 5.9996e+00
```

# 递归神经网络的应用

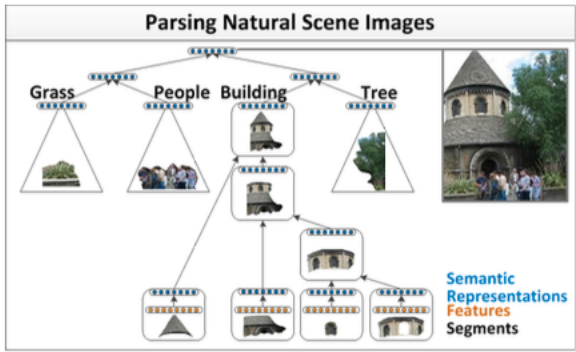
## 自然语言和自然场景解析

在自然语言处理任务中，如果我们能够实现一个解析器，将自然语言解析为语法树，那么毫无疑问，这将大大提升我们对自然语言的处理能力。解析器如下所示：



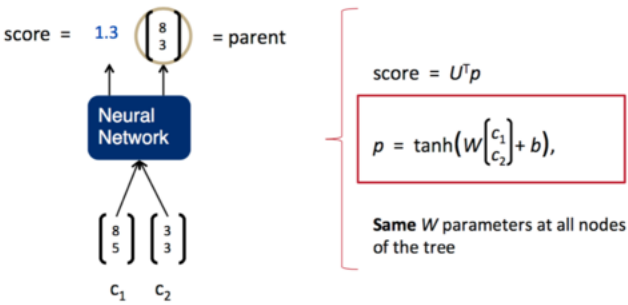
可以看出，递归神经网络能够完成句子的语法分析，并产生一个语法解析树。

除了自然语言之外，自然场景也具有可组合的性质。因此，我们可以用类似的模型完成自然场景的解析，如下图所示：

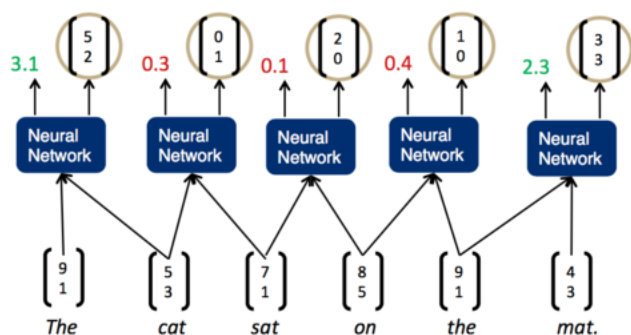


两种不同的场景，可以用相同的递归神经网络模型来实现。我们以第一个场景，自然语言解析为例。

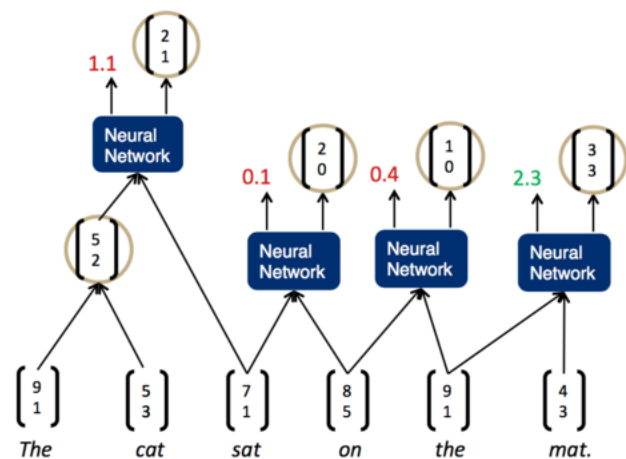
我们希望将一句话逐字输入到神经网络中，然后，神经网络返回一个解析好的树。为了做到这一点，我们需要给神经网络再加上一层，负责打分。分数越高，说明两个子节点结合更加紧密，分数越低，说明两个子节点结合更松散。如下图所示：



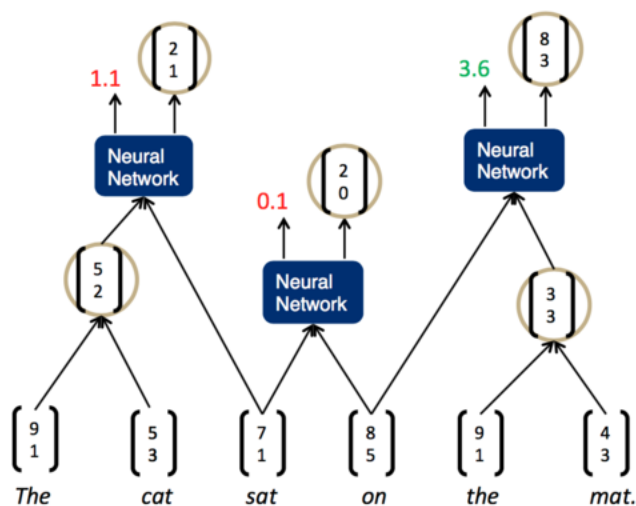
一旦这个打分函数训练好了（也就是矩阵U的各项值变为合适的值），我们就可以利用贪心算法来实现句子的解析。第一步，我们先将词按照顺序两两输入神经网络，得到第一组打分：



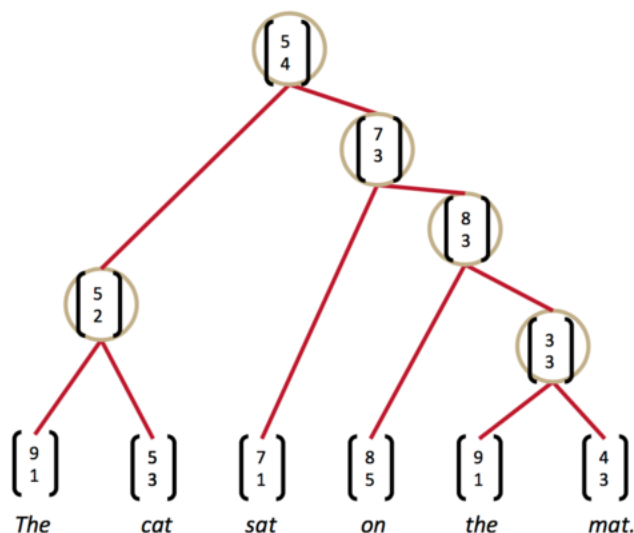
我们发现，现在分数最高的是第一组，The cat，说明它们的结合是最紧密的。这样，我们可以先将它们组合为一个节点。然后，再次两两计算相邻子节点的打分：



现在，分数最高的是最后一组，the mat.。于是，我们将它们组合为一个节点，再两两计算相邻节点的打分：



这时，我们发现最高的分数是on the mat，把它们组合为一个节点，继续两两计算相邻节点的打分.....最终，我们就能够得到整个解析树：



现在，我们困惑这样牛逼的打分函数score是怎样训练出来的呢？我们需要定义一个目标函数。这里，我们使用Max-Margin目标函数。它的定义如下：

$$J(\theta) = \max(0, \sum_i \max_{y \in A(x_i)} (s(x_i, y) + \Delta(y, y_i)) - s(x_i, y_i))$$

在上式中， $x_i$ 、 $y_i$ 分别表示第*i*个训练样本的输入和标签，注意这里的标签 $y_i$ 是一棵解析树。 $s(x_i, y_i)$ 就是打分函数s对第*i*个训练样本的打分。因为训练样本的标签肯定是正确的，我们希望s对它的打分越高越好，也就是 $s(x_i, y_i)$ 越大越好。 $A(x_1)$ 是所有可能的解析树的集合，而 $s(x_i, y)$ 则是对某个可能的解析树y的打分。 $\Delta(y, y_i)$ 是对错误的惩罚。也就是说，如果某个解析树y和标签 $y_i$ 是一样的，那么 $\Delta(y, y_i)$ 为0，如果网络的输出错的越离谱，那么惩罚项 $\Delta(y, y_i)$ 的值就越高。 $\max(s(x_i, y) + \Delta(y, y_i))$ 表示所有树里面最高得分。在这里，惩罚项相当于Margin，也就是我们虽然希望打分函数s对正确的树打分比对错误的树打分高，但也不要高过Margin的值。我们优化 $\theta$ ，使目标函数取最小值，即：

$$\theta = \underset{\theta}{\operatorname{argmin}} J(\theta)$$

下面是惩罚函数 $\Delta$ 的定义：

$$\Delta(y, y_i) = k \sum_{d \in N(y)} \mathbf{1}\{\text{subTree}(d) \notin y_i\}$$

上式中， $N(y)$ 是树y节点的集合；subTree(d)是以d为节点的子树。上式的含义是，如果以d为节点的子树没有出现在标签 $y_i$ 中，那么函数值+1。最终，惩罚函数的值，是树y中没有出现在树 $y_i$ 中的子树的个数，再乘上一个系数k。其实也就是关于两棵树差异的一个度量。

$s(x, y)$ 是对一个样本最终的打分，它是对树y每个节点打分的总和。

$$s(x, y) = \sum_{n \in \text{nodes}(y)} s_n$$

具体细节，读者可以查阅『参考资料3』的论文。

## 小结

我们在系列文章中已经介绍的全连接神经网络、卷积神经网络、循环神经网络和递归神经网络，在训练时都使用了监督学习(Supervised Learning)作为训练方法。在监督学习中，每个训练样本既包括输入特征 $x$ ，也包括标记 $y$ ，即样本 $d^{(i)} = \{x^{(i)}, y^{(i)}\}$ 。然而，很多情况下，我们无法获得形如 $\{x^{(i)}, y^{(i)}\}$ 的样本，这时，我们就不能采用监督学习的方法。在接下来的几篇文章中，我们重点介绍另外一种学习方法：增强学习(Reinforcement Learning)。在了解增强学习的主要算法之后，我们还将介绍著名的围棋软件AlphaGo，它是一个把监督学习和增强学习进行完美结合的案例。





## 参考资料

1. [CS224d: Deep Learning for Natural Language Processing](#)
2. [Learning Task-Dependent Distributed Representations by Back Propagation Through Structure](#)
3. [Parsing Natural Scenes and Natural Language with Recursive Neural Networks](#)

## · 内容目录

### 。 [零基础入门深度学习\(7\) - 递归神经网络](#)

- [文章列表](#)
- [往期回顾](#)
- [递归神经网络是啥](#)
- [递归神经网络的前向计算](#)
- [递归神经网络的训练](#)
  - [误差项的传递](#)
  - [权重梯度的计算](#)
  - [权重更新](#)
- [递归神经网络的实现](#)
- [递归神经网络的应用](#)
  - [自然语言和自然场景解析](#)
- [小结](#)
- [参考资料](#)

·

- - 。 机器学习 7
  - [零基础入门深度学习\(7\) - 递归神经网络](#)
  - [零基础入门深度学习\(6\) - 长短期记忆网络\(LSTM\)](#)
  - [零基础入门深度学习\(5\) - 循环神经网络](#)
  - [零基础入门深度学习\(4\) - 卷积神经网络](#)
  - [零基础入门深度学习\(3\) - 神经网络和反向传播算法](#)
  - [零基础入门深度学习\(2\) - 线性单元和梯度下降](#)
  - [零基础入门深度学习\(1\) - 感知器](#)
  - 。 深度学习入门 7

- [零基础入门深度学习\(7\) - 递归神经网络](#)
- [零基础入门深度学习\(6\) - 长短期记忆网络\(LSTM\)](#)
- [零基础入门深度学习\(5\) - 循环神经网络](#)
- [零基础入门深度学习\(4\) - 卷积神经网络](#)
- [零基础入门深度学习\(3\) - 神经网络和反向传播算法](#)
- [零基础入门深度学习\(2\) - 线性单元和梯度下降](#)
- [零基础入门深度学习\(1\) - 感知器](#)

◦

◦ 以下【标签】将用于标记这篇文稿：

- 
- 
- 
- - [下载客户端](#)
  - [关注开发者](#)
  - [报告问题，建议](#)
  - [联系我们](#)
- 

添加新批注



保存

取消

在作者公开此批注前，只有你和作者可见。



保存

取消



修改

保存

取消

删除

- 私有
- 公开
- 删除

查看更早的 5 条回复

回复批注



通知

取消

确认

- ☐
- ☐